VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# GENERÁTOR DOKUMENTACE PRO TESTY POUŽÍVAJÍCÍ KNIHOVNU BEAKERLIB

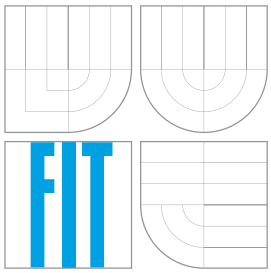
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KULDA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# GENERÁTOR DOKUMENTACE PRO TESTY POUŽÍVAJÍCÍ KNIHOVNU BEAKERLIB

AUTOMATED TEST DOCUMENTATION GENERATOR FOR BEAKERLIB TESTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KULDA

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2015

# Abstrakt

Cílem této práce řešené ve spolupráci se společností Red Hat Czech je navrhnout, implementovat a ověřit generátor dokumentace pro testy používající knihovnu BeakerLib, který efektivně vytváří dokumentaci z neokomentovaných BeakerLib testů. V prvním kroku generátor extrahuje data z BeakerLib příkazů. Následně jsou data přetvořena do informací v přirozeném jazyce. Na závěr jsou tyto informace vloženy do šablony dokumentace. Při tvorbě generátoru dokumentace byl použit modul argparse pro hledání dat z BeakerLib příkazů. Ve srovnání s existujícími nástroji navržený generátor přináší nový způsob vytváření dokumentací bez použití dokumentačních komentářů. Díky této vlastnosti lze generovat dokumentace, které jsou vytvořeny na základě automatizovaného porozumění zdrojového kódu testu. Testování, po celou dobu vývoje generátoru, probíhalo na třech zvolených BekerLib testech. Na závěr byl generátor otestován na deseti náhodně zvolených BeakerLibových testech.



# Abstract

The aim of this work in cooperation with Red Hat Czech company is to design, implement and verify documentation generator for test written using BeakerLib library, which effectively creates documentation from BeakerLib tests without any documentation markup. In the first step generator parses data from every BeakerLib command in the test. Subsequently data are transformed as a natural language information. At the end generator transforms this information into documentation template. In this case an argparse method was used to find possible data from BeakerLib commands. In contrast to existing documentation generators this generator brings a new way of documentary creation from tests without any documentation markup. Thanks to this point of view we can generate documentation, which is created on base of automated understanding of test source code. Through documentation generator development time the generator was tested on three BeakerLib tests In the end the documentation generator was tested on ten BeakerLib tests which were randomly selected.



# Klíčová slova

BeakerLib, generátor dokumentace, python, shlex, argparse, shellové testy, problém batohu

# Keywords

BeakerLib, documentation generator, python, shlex, argparse, shell tests, knapsack problem

# Citace

Jiří Kulda: Generátor dokumentace pro testy používající knihovnu BeakerLib, bakalářská práce, Brno, FIT VUT v Brně, 2015

# Generátor dokumentace pro testy používající knihovnu BeakerLib

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Vojnara, Ph.D. a Ing. Petra Müllera. Informace k práci jsem čerpal ze zdrojů uvedených v seznamu literatury.

.....
Jiří Kulda
19. května 2015


## Poděkování

Rád bych touto cestou poděkoval mému konzultantovi z firmy Red Hat Czech Ing. Petru Müllerovi, který mě vedl k dosažení cíle po odborné stránce a dělil se se mnou o své znalosti z oblasti testování. Dále bych rád poděkoval vedoucímu práce prof. Ing. Tomáši Vojnarovi, Ph.D., který práci vedl na straně FIT VUT a to zejména po pedagogické stránce. Na závěr bych rád poděkoval panu Mgr. Davidu Kutálkovi z firmy Red Hat Czech za poskytnutí zpětné vazby k vytvořeným dokumentacím z vybraných testů.

# Obsah

# Kapitola 1

# Úvod

Cílem této práce je řešit problém, jak efektivně vytvářet z kódu testu dokumentaci v přirozeném jazyce. V dnešní době existuje velký objem dokumentace, která neodpovídá programu, který dokumentuje. K této situaci dochází zejména proto, že se k první verzi programu vytvoří dokumentace a postupem času, kdy se program mění, tak dokumentace zůstává stále stejná. Tento problém by automatizované generování dokumentace mělo odstranit. V této práci se pak konkrétně zaměříme na návrh, implementaci a otestování generátoru.

V následujícím textu 2 jsou nejprve představeny technologie, na nichž tato práce staví. Následně je diskutován současný stav v oblasti automatizace generování dokumentace a je představena naše vlastní představa o dokumentaci k *BeakerLib* testům, podložená diskuzí s tvůrci těchto testů z firmy Red Hat Czech.

Implementace celého generátoru je popsána v kapitole 4. Důraz je kladen na získání podstatných dat či faktů z kódu testu, vytváření informačních jednotek a jejich logické spojování do komplexnějších informací.

V předposlední kapitole jsou porovnány tři ručně vytvořené dokumentace s vygenerovanými dokumentacemi. Následně generátor je otestován na deseti zvolených *BeakerLib* testech a k těmto testům je poskytnuta informace o přínosu vygenerovaných dokumentací.

Poslední kapitola se zabývá shrnutím vytvořeného generátoru a jsou prezentovány možnosti dalšího vývoje.

# Kapitola 2

# BeakerLib knihovna a testy

Tato kapitola slouží k seznámení s knihovnou *BeakerLib* a s testy z této knihovny vycházejícími. Pochopení této kapitoly je velmi důležité pro porozumění dalším částem této práce.

## 2.1 BeakerLib knihovna

Jedná se o knihovnu vytvořenou v *shellu* primárně pro tvorbu testů [7]. *BeakerLib* poskytuje komfortní funkce, které usnadňují práci při tvorbě testů, spouštění a tvorby *blackbox* testů. Mezi hlavní rysy této knihovny patří:

- *Journal*. Základem pro fungování *BeakerLib* knihovny je jednoduchý logovací mechanismus, který vytváří tzv. žurnál v *XML* formátu. Žurnál uchovává informace o každém spuštěném *BeakerLib* příkazu. Jde o jednotný logovací mechanismus.
- *Phases*. Jsou definovány standardní fáze testů. V rámci jednotlivých fází jsou použity příkazy, jejichž funkčnost odpovídá činnosti, kterou daná fáze provádí. Příklady takových fází a příkazů budou uvedeny níže.
- *Asserts*. Podpora běžných kontrol. *BeakerLib* knihovna poskytuje velkou škálu příkazů pro kontrolu jednotlivých požadavků při tvorbě testů. Může se jednat o kontrolu návratových kódů, existence souborů, porovnávání hodnot a mnoho dalších.
- *Helpers*. *BeakerLib* dále poskytuje kolekci pomocných funkcí tzv. *Helpers*. Například existuje příkaz `rlPerfTime_RunsInTime`, který ověřuje kolikrát se může spustit zvolený příkaz v určitém čase.

## 2.2 BeakerLib test

Na ukázce kódu 2.1 můžeme vidět jednoduchý *BeakerLib* test. Tyto testy vypadají hodně odlišně než klasické testy v *shellu*. Jak bylo uvedeno již výše, testy jsou rozčleněny do fází. Na ukázce 2.1 se nachází fáze *Setup*, dvě fáze *Test* a na konci fáze *Cleanup*. Využívání těchto fází je velmi užitečné, protože již název fáze naznačuje, jaké příkazy bude fáze spouštět. Fáze *Cleanup* bude například obsahovat pouze příkazy související s úklidem po testu a podobně tomu bude i u ostatních fází. Následně na ukázce kódu můžeme vidět příkazy začínající na souhlásky "rl". Jedná se o *BeakerLib* příkazy usnadňující tvorbu testů přičemž mezi nejpoužívanější příkazy patří:

- `rlRun`. Tento příkaz očekává jako první parametr příkaz a následně návratový kód tohoto příkazu. Jedná se tedy o kombinaci dvou funkcí. Příkaz `rlRun` spouští zvolený příkaz a následně kontroluje návratový kód spuštěného příkazu, zda-li odpovídá zvolenému návratovému kódu. Pokud budou rozdílné, tak příkaz i celá fáze skončí neúspěchem. Jedná se o nejdůležitější a nejpoužívanější *BeakerLib* příkaz.

- `rlFileBackup`. Slouží pro tvorbu zálohy pro daný soubor nebo adresář. U tohoto příkazu se mnohdy využívá parametru `--clean`, který rekurzivně odstraňuje změny v daném adresáři těsně před obnovením adresáře použitím příkazu `rlFileRestore`.

- `rlServiceStart`. Slouží pro spuštění zvolené služby. Dále při prvním spuštění se uloží momentální stav služby, aby bylo jednoduché vrátit službu do předcházejícího nastavení.

- `rlAssertExists`. Jedná se o kontrolu, zda zvolený soubor či adresář existuje. Příkaz vrací nulu, pokud daný příkaz či adresář opravdu existuje.

- `rlAssertGrep`. Jedná se o kontrolu, zda zvolený soubor obsahuje zvolený řetězec. Dále můžeme využít rozšiřujících parametrů `-i`, který nerozlišuje malá a velká písmena, nebo `-E`, který poskytuje rozšířená vyhledávání.

- `rlWatchdog`. Tento příkaz očekává jako první parametr příkaz a následně časový limit příkazu. Příkaz `rlWatchdog` se tedy používá pro příkazy, které mohou běžet příliš dlouho a chceme jejich dobu omezit určitým časovým limitem. Pokud dojde k překročení časového limitu, tak daný příkaz bude ukončen pomocí příslušného signálu.

- `rlAssertRpm`. Tento příkaz zajišťuje, že zvolený balíček je nainstalován a další příkazy ho mohou používat. U tohoto příkazu lze nastavit i příslušnou verzi, vydání nebo architekturu balíčku.

Výše uvedené příkazy jsou nicméně jen malou ukázkou. *BeakerLib* poskytuje více jak šedesát příkazů, představujících velmi širokou škálu příkazů pro pohodlnější tvorbu testů. Navíc, pokud bychom chtěli do testu vložit vlastní *shellové* příkazy, tak je zde i tato možnost, protože *BeakerLib* je *shellová* knihovna, která tyto příkazy podporuje. Mimo jiné můžeme vytvářet klasické funkce a cykly známé z *shellu*.

```
rlJournal Start
  rlPhaseStartSetup "Setup"
    rlAssertRpm "httpd"
    rlRun 'TmpDir=$(mktemp -d)' 0
    pushd $TmpDir
    rlRun "rlFileBackup --clean $HttpdPages $HttpdLogs" 0 "Backing
       up"
    rlRun "echo 'Welcome to Test Page!' > $HttpdPages/index.html" 0
       "Creating a simple welcome page"
    rlRun "rm -f $HttpdLogs/*"
    rlRun "rlServiceStart httpd"
  rlPhaseEnd

  HttpdPages="/var/www/"
  rlPhaseStartTest "Test Existing Page"
    rlRun "wget http://localhost/" 0 "Fetching the welcome page"
    rlAssertExists "index.html"
    rlLog "index.html contains: $(<index.html)"
    rlAssertGrep "Welcome to Test Page" "index.html"
    rlAssertGrep "GET / HTTP.*200" "$HttpdLogs/access_log"
  rlPhaseEnd

  rlPhaseStartTest "Test Missing Page"
    rlRun "wget http://localhost/missing.html 2>stderr" 1,8 \
      "Trying to access a nonexistent page"
    rlAssertNotExists "missing.html"
    rlAssertGrep "Not Found" "stderr"
    rlAssertGrep "GET /missing.html HTTP.*404" "$HttpdLogs/
       access_log"
    rlAssertGrep "does not exist.*missing.html" "$HttpdLogs/
       error_log"
  rlPhaseEnd

  rlPhaseStartCleanup "Cleanup"
    popd
    rlRun "rm -r TmpDir" 0 "Removing tmp directory"
    rlRun "rlFileRestore"
    rlRun "rlServiceRestore httpd"
  rlPhaseEnd
rlJournalEnd
```

Kód 2.1: Příklad jednoduchého BeakerLibového testu

# Kapitola 3

# Návrh dokumentace

První část této kapitoly vysvětluje pojmy *statická analýza* a *dynamická analýza*. Následně poskytuje přehled již existujících dokumentačních nástrojů. V další části je navržen formát dokumentace, který se jeví jako ideální na základě řady diskuzí s vývojáři firmy Red Hat Czech.

## 3.1 Statická a dynamická analýza programu

Statická analýza je důsledné zkoumání zdrojového kódu programu bez jeho spuštění. Tato analýza se využívá pro hledání chyb v kódu. Typickým příkladem je překladač, který hledá lexikální, sémantické a syntaktické chyby v kódu. Statická analýza se také používá pro vylepšení kvality kódu. V dnešní době většina generátorů dokumentace používá statickou analýzu.

Dynamická analýza je analýza programu, která se provádí v době běhu programu. Tato analýza může odhalit další chyby, které nelze vyřešit pomocí statické analýzy. Dále poskytuje užitečnou informaci programátorovi o chování programu. Pomocí dynamické analýzy můžeme sledovat volání knihoven za běhu programu.

## 3.2 Existující nástroje

### 3.2.1 Javadoc

Jeden z nejznámějších a nejrozšířenějších nástrojů pro podporu tvorby dokumentace je *Javadoc* [8], který je vytvořen pouze pro programovací jazyk *Java*, jak už jeho název napovídá. Tento nástroj generuje výstupní dokumentaci v *HTML* formátu z dokumentačních komentářů v zdrojovém kódu.

Dokumentační komentář obsahuje specificky naformátované místa, která se vyznačují znakem @ na začátku slova a takto označené slovo se nazývá *tag*. Jde tedy o generování dokumentace ze specificky naformátovaných komentářů a pro tento nástroj se používá formát *javadoc*. S označenými komentáři se můžeme často setkat i v různých programovacích jazycích jako jsou například jazyky *C* nebo *C++* a mnoho dalších. Mezi nejpoužívanější *tagy* patří např.:

- *@author jméno*. Přidá jméno autora do vygenerované dokumentace.
- *@param parametr popis*. Přidá parametr metody s popisem do vygenerované dokumentace.

- *@return popis.* Přidá popis návratové hodnoty do vygenerované dokumentace.
- *@version verze.* Přidá verzi programu do vygenerované dokumentace.

Označené komentáře zvyšují čitelnost celého zdrojového kódu, protože zásluhou *tagů* lze vytvářet strukturované komentáře s popisy důležitých míst. Jak už bylo zmíněno dříve, výstupním formátem je *HTML*, tak lze použít *tagy* vytvořené pro tento jazyk a více tak zvýraznit jednotlivé části dokumentace. Pokud bychom chtěli zvýraznit jednotlivé části komentářů, například bychom chtěli některé části tučně, tak musíme do komentářů vložit ještě *HTML tagy*, což způsobí větší nesrozumitelnost komentáře. Na ukázce 1 lze vidět rozdíl mezi komentáři u dvou příkladů. Komentáře jsou napsány v *Javadoc* formátu s tím rozdílem, že v pravé části byly navíc použity *HTML tagy* pro změnění výstupu. Můžeme si všimnout, že díky přidaným *HTML tagům* klesá čitelnost tohoto komentáře. Lepší čitelnosti lze dosáhnout u pokročilejších vývojových prostředí umožňující zobrazovat náhledy komentářů s již zvýrazněnými úseky.

- *Výhody:* Generování stabilní dokumentace v *HTML* podobě.
- *Nevýhody:* Neexistuje pro více programovacích jazyků. Nízká automatizace, jedná se především o přesázení textu do přehlednější šablony.

```
/**
* This method prints Hello World
* with specified name
*
* @param name is string with speci-
* fied name
* @return succes of this method
*/
public int print_hello(string name)
{
  string pom = "Hello World " + name
      ;
  System.out.println(pom);
  return 0;
}
```

```
/**
* This method prints <b>Hello
    World<\b>
* with specified <u>name<\u>
*
* @param name is string with
   specified name
* @return <b>succes<\b> of
   this method
*/
public int print_hello(string
    name)
{
  string pom = "Hello World "
      + name;
  System.out.println(pom);
  return 0;
}
```

Kód 1: Ukázka komentářů v Javadoc formátu s tagy.

### 3.2.2 Doxygen

*Doxygen* [3] také patří mezi jedny z nejznámějších a nejpoužívanějších nástrojů. Jedná se o nástroj generující dokumentaci z okomentovaného *C++* kódu. Nicméně tento nástroj dokáže generovat dokumentaci i z různých jazyků, například: *C#*, *Python*, *VHDL*, *PHP* a mnoho dalších. Kromě formátu komentářů *Javadoc*, *Doxygen* podporuje dokumentační tagy používané v *QT* nástroji. Výstupním formátem může být opět *HTML*, nicméně *Doxygen* podporuje i další formáty mezi nejznámější patří například: *RTF*, *PDF* a *PostScript*.

```
/**
* Function: Multiply
*
* Multiplies two integers
*
* Parameters:
*    x - The first integer
*    y - The second integer
*
* Returns:
*   The two integers multiplied
*   together
*
* See Also:
*   <Divide>
*/
int Multiply(int x, int y)|
{
  return x * y;
}
```

```
/**
* Multiplies two integers.
*
*
* @param x The first integer.
* @param y The second integer.
* @return The two integers
*      multiplied together
* @see Divide
*/
int Multiply(int x, int y)
{
   return x * y;
}
```

Kód 2: Ukázka komentářů ve formátu Natural Docs (levý sloupec) a Javadoc (pravý sloupec)

Tento nástroj umí navíc automaticky generovat grafy závislostí a grafy dědičnosti pro lepší zobrazení vztahů mezi jednotlivými prvky.

- *Výhody:* Možnost volby více výstupních formátů dokumentace. Podporuje *Javadoc* formát. Automaticky vytváří více (grafy závislostí).
- *Nevýhody:* Generování dokumentace závislé na vytvořených komentářích. Trochu zvýšená automatizace, ale pořád je na velmi nízké úrovni.

### 3.2.3 Natural Docs

Mezi méně známé nástroje patří *Natural Docs* [10]. Opět se jedná o nástroj generující dokumentaci z okomentovaného kódu, ale snaží se, aby výstupní dokumentace vypadala, jako by ji napsal člověk, ne stroj. Tento nástroj podporuje generování dokumentace z devatenácti jazyků. Mezi nejznámější patří *C++*, *Java* a *Python*. Nicméně *Natural Docs* používá svůj vlastní dokumentační formát namísto známého *Javadoc* formátu. Na ukázce kódu 2 můžeme vidět rozdíl mezi *Natural Docs* a *Javadoc* formátem. Můžeme vidět, že *Natural Docs* formát je více strukturovaný a čitelnější než *Javadoc* formát. Díky větší strukturovanosti, komentář zabírá daleko více místa v kódu a přispívá k větší míře rolování kódu.

- *Výhody:* Podporuje více programovacích jazyků.
- *Nevýhody:* Výstup pouze v *HTML* podobě. Nepodporuje *Javadoc* formát. Nízká automatizace, jedná se především o přesázení textu do přehlednější šablony.

### 3.2.4 Automatický generátor dokumentace přes sumarizaci zdrojového kódu

Předchozí přístupy nenabízely vysoký stupeň automatizace při generování komentářů. S tímto nedostatkem se snaží vyrovnat např. práce [5]. Tato práce se zabývá generováním dokumentace na základě porozumění vstupního kódu. Tento nástroj je primárně vytvořen pro jazyk *Java* a získává informace ze jmen metod a jejich kódu. Následně využívá podpory tří technologií:

- *Software Word Usage Model.* Tato technika slouží k reprezentaci příkazů programu jako souboru sloves, podstatných jmen a předložkových vazeb.
- *Nature Language Generation System.* Metoda pro generování přirozeného jazyka. Tato metoda přijímá data a díky pokročilé znalosti anglického jazyka poté vytváří anglické věty, které obsahují obdržená data.
- *PageRank.* Jedná se o užitečný algoritmus pro ohodnocení důležitosti webových stránek. Nicméně se začíná více rozšiřovat do softwarového inženýrství [4, 1, 6]. Zde se například používá pro zvýraznění důležitých funkcí nebo metod v programu.

Postup při získávání informací ze zdrojového kódu je následující. Nejprve se pomocí technologie *PageRank* naleznou nejvíce důležité metody. Následně se využívá technologie *Software Word Usage Model* na získání klíčových slov a akcí, které tyto metody mají provést. Na závěr se využije technologie *Nature Language Generation System* pro generování anglických vět pro popis použití daných metod.

- *Výhody:* Generování dokumentace na základě porozumění zdrojového kódu. Vyšší automatizace. Nejedná se jenom o přeformátování slov do šablon, ale o pokročilou extrakci dat z kódu programu a tvorbu anglických vět v lidské řeči, které jsou vloženy do dokumentace.
- *Nevýhody:* Pouze pro programovací jazyk *Java*. Nicméně použité metody lze použít i v jiných programovacích jazycích.

### 3.2.5 Shrnutí existujících nástrojů

Vyjmenovány zde byly především jedny z nejznámějších dokumentačních nástrojů. Tyto nástroje poskytují velkou škálu funkcí. Ve většině nástrojů si můžeme vybrat libovolný výstupní formát nebo podporují větší počet programovacích jazyků, ze kterých generují dokumentaci. Nicméně první tři nástroje vytvářejí dokumentaci na základě okomentovaného kódu. Jedná se o velký nedostatek, protože jakákoli úprava ve zdrojovém kódu, může změnit popisovanou funkci a následně se musí přepracovat komentář k této funkci. Tato vlastnost vede k tomu, že v dnešní době je spousta dokumentací, které přesně nemusí odpovídat popisovanému kódu. Nový způsob přináší poslední zmiňovaný nástroj 3.2.4. Tato vědecká práce popisuje cestu generování dokumentace ze zdrojového kódu. Jedná se o nový způsob, který přináší mnoho výhod. Největší výhodou je generování přesné dokumentace a udržování její aktuálnosti.

V dnešní době existuje spousta dokumentačních nástrojů. Bohužel většina z nich stále generuje dokumentaci pouze ze specificky označených komentářů. Změnu přináší vědecká práce, která popisuje cestu generování dokumentace z porozumění vstupního kódu a tato cesta bude vysvětlena dále v této práci.

## 3.3 Formát dokumentace pro BeakerLib testy

V této sekci bude vysvětlen průběh tvorby návrhu dokumentace pro *BeakerLib* testy. Následně je zde zmíněno, co za informace nesmí v dokumentaci chybět a také, jaké má být rozvržení jednotlivých informací v dokumentaci.

### 3.3.1 Proces tvorby dokumentace

Nejprve bylo důležité si uvědomit, jak by měla dokumentace vypadat a co za informace by měla obsahovat. Dokumentace by měla poskytovat užitečné informace a hlavně ve strukturované podobě. Pro vytvoření prvního návrhu dokumentace byly vybrány tři *BeakerLib* testy (apache, mozilla, pxe-boot), ke kterým byla ručně vytvořena dokumentace.

Pro ověření, zda vytvořený návrh dokumentace je použitelný v praxi, bylo zorganizováno setkání s testery ze společnosti Red Hat Czech.

Od tohoto setkání byly očekávány informace ohledně vzhledu ideální dokumentace, získání důležitých dat z příkazů *BeakerLib* a informací o nástrojích, které využívají v již existujících dokumentačních generátorech. Mezi otázky patřilo například: „Jak má být ideální dokumentace velká“ nebo „jak získat z jednotlivých příkazů co nejvíce informací“. Z debaty vyplynulo několik velmi zajímavých poznatků, například:

- Ideální dokumentace neexistuje, každý tester by rád měl dokumentaci šitou na míru jeho potřebám.
- Dokumentace může být extrémně dlouhá, pokud bude ve strukturovaném formátu.
- Dokumentace by mohla zobrazovat informace o chybách, ke kterým může dojít při spuštění testu a doporučení, jak tyto chyby opravit.
- Dokumentace by mohla být připojena k verzovacímu nástroji, aby byly vidět změny v dokumentaci, při změnách v testu.

Dokumentace by měla být šitá na míru, zobrazovat chyby, ke kterým může dojít při spuštění. Z výše uvedené debaty tedy plyne, že je zapotřebí nějaký inteligentní generátor, který se bude přizpůsobovat.

V průběhu debaty byly položeny otázky, zda první verze formátu dokumentace odpovídá potřebám pro práci testerů a zda obsahuje všechny důležité informace. Testerům se až na malé připomínky formát líbil, přestože připomínky testerů byly malé, vytvořený dokumentační formát byl ještě vylepšen. Toto vylepšení je rozebíráno následující části.

### 3.3.2 Finální návrh dokumentace

Finální návrh dokumentace je rozdělen do tří částí.

První část je určena pro rychlé seznámení čtenáře s obsahem dokumentace. Musí být umístěna ve vrchní části dokumentace, aby byly tyto informace co nejrychleji přístupné čtenáři. V této části jsou obsažena data o autorovi testu (jméno autora, email atd.). Následně zde nesmí chybět souhrnná informace o testu, která ve dvou větách vysvětlí podstatu testu. Další informací jsou klíčová slova, která primárně slouží pro rychlé vyhledávání mezi dokumentacemi. Pod klíčovými slovy se nachází informace o struktuře testu. Díky struktuře testu se čtenář dozví kolik test obsahuje *BeakerLib* fází. Na konci této části nalezneme informaci o spuštění testu. Tato informace napovídá čtenáři, jak spustit patřičný test a kolik parametrů potřebuje pro správné spuštění.

Druhá část, která je umístěna v prostřední části dokumentace, obsahuje informace o testu. Informace v této části jsou rozděleny do bloků. Jeden blok odpovídá jedné *BeakerLib* fázi. Každý blok obsahuje nadpis a ihned pod tímto nadpisem jsou informace o činnosti dané fáze. Nadpis bloku se skládá z názvu fáze a informace o důvěryhodnosti fáze. Informace o důvěryhodnosti je velmi důležitá z důvodu reflexe, jak dokumentace odpovídá fázi, ze které vychází. Jedná se o to, že generátor nemusí rozpoznat všechny příkazy a měl by poskytovat informaci, z kolika příkazů byla dokumentace vygenerována. Díky tomuto poznatku bude čtenář vědět, jak moc má vygenerované dokumentaci důvěřovat. Informace o důvěryhodnosti chyběla v prvním návrhu a byla přidána díky zmiňované zpětné vazbě.

Třetí část obsahuje dodatečné informace o testu. Tato část je umístěna pod druhou částí dokumentace. V této části můžeme naleznout dodatečné informace o funkcích, cyklech a podmínkách obsažených v testu. Dále zde můžeme najít informaci o předpokládaném výsledku testu. Generátor spustí daný test a jeho výsledky poté uloží do této části.

Dokumentace 1 zobrazuje přesnou finální verzi návrhu dokumentace pro *apache* test. Můžeme vidět, že tato dokumentace odpovídá popisovanému návrhu.

```
Description:  Apache example test
Author:  Petr Splichal <psplicha@redhat.com>
Keywords: apache, httpd

Test structure: Setup, 2xTest, Cleanup

Test launch: ./apache-test.sh

Setup: [Unknown commands = 3, Total = 7]
   Package httpd must be installed
   Service httpd must be running

Test1: [Unknown commands = 3, Total = 5]
   Command wget must run succesfuly
   File index.html must exists and contain string "Welcome to Test Page"
   File /var/log/httpd/access_log must contain string "GET / HTTP.*200"

Test2: [Unknown commands = 3, Total = 5]
   Command wget must not run succesfuly and it's stderr must contain
        string "Not Found"
   File "missing.html" must not exist
   File access_log must contain string "GET /missing.html HTTP.*404"
   File error_log must contain string "does not exist.*missing.html"

Cleanup: [Unknown commands = 2, Total = 4]
   Restore state of httpd service

Expected result:
```

Dokumentace 1: Finální návrh dokumentace pro Apache test.

# Kapitola 4

# Analýza zdrojového kódu a generování dokumentace

V této kapitole bude vysvětlen postup analýzy zdrojového kódu a získávání dat z jednotlivých příkazů. Následně bude ukázáno, jak tato data transformovat do informace v přirozené lidské řeči. Na závěr bude vysvětlen průběh vypisování těchto informací.

## 4.1 Získání dat ze zdrojového kódu

Tato sekce se zabývá procesem získání důležitých dat z testových příkazů. V tomto kroku generátor využívá moduly *argparse* a *shlex*, které jsou primárně vytvořeny pro programovací jazyk *Python*.

### 4.1.1 Modul shlex

Jedná se o jednoduchý lexikální analyzátor s klasickými metodami jako `get_token()`. Z názvu této metody můžeme očekávat, že vrací *tokeny*, nicméně se jedná o lexémy. Rozdíl mezi *tokenem* a *lexémou* je ten, že *lexéma* je lexikální jednotka daného programovacího jazyka, např. identifikátor, celočíselná konstanta, operátor sčítání, apod. *Token* je konkrétní reprezentace lexémy, např. identifikátor *fox* či celočíselná konstanta *6*[1]. Navíc tento modul poskytuje velkou škálu funkcí, jako například metodu `put_token(item)` pro vrácení hodnoty *item* zpět na zásobník, metodu `error_leader()` pro generování chybových hlášení a mnoho dalších.

Tento modul není jen jednoduchý lexikální analyzátor, ale i výborný rozdělovač vět do pole slov podle bílých mezer. U tohoto rozdělovače lze nastavit, zda-li má odstranit případné komentáře z věty, kterou rozděluje. Tato vlastnost je využita k odstranění nežádoucích komentářů z příkazů. Dále lze nastavit výstupní formát rozdělené věty. Můžeme si vybrat mezi dvěma formáty *POSIX* a *non-POSIX*. Na obrázku 4.1 můžeme vidět, že formát *POSIX* odstraňuje znaky uvozovek a také zpětná lomítka, která nejsou zahrnuta v uvozovkách. Této vlastnosti je využito pro nastavení korektního vstupu pro modul *argparse*.

### 4.1.2 Modul argparse

Tento modul byl přidán do programovacího jazyka *Python 2.7* jako náhrada za starší *optparse*, který je velmi podobný známějšímu *getopt* z programovacího jazyku *C*. Jedná se

[1] http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/

```
ORIGINAL: 'Escaped "\\e" Character \\in double quotes'
non-POSIX: ['Escaped', '"\\e"', 'Character', "'\\in'", 'double',
        'quotes']
POSIX: ['Escaped', '\\e', 'Character', 'in', 'double', 'quotes']
```

Kód 4.1: Ukázka formátu rozdělovače.

o překladač pro argumenty příkazové řádky a dílčí příkazy. Modul *argparse* se primárně využívá k získání všech argumentů příkazové řádky. Nicméně v této práci je modul využit pro získávání dat z *BeakerLib* příkazů. Hlavní vlastnost tohoto modulu je, že se dá snadno upravit tak, aby získal jakékoli argumenty. V modulu lze nastavit opakování argumentů, jejich datový typ, úložiště nebo výchozí hodnotu pro daný argument. Výhodou tohoto modulu je, že automaticky generuje nápovědu, informaci o použití a pokud dostane špatné parametry, tak tiskne chybová hlášení. Díky těmto vlastnostem napovídá autorovi, kde se v testu mohla stát chyba a jak ji napravit. Na ukázce kódu 4.2 lze vidět nastavení *argparse* modulu pro *rlRun* příkaz.

```
parser_arg = argparse.ArgumentParser()
parser_arg.add_argument("argname", type=str)
parser_arg.add_argument('-t', dest='t', action='store_true', default
    =False)
parser_arg.add_argument('-l', dest='l', action='store_true', default
    =False)
parser_arg.add_argument('-c', dest='c', action='store_true', default
    =False)
parser_arg.add_argument('-s', dest='s', action='store_true', default
    =False)
parser_arg.add_argument("command", type=str)
parser_arg.add_argument("status", type=str, nargs='?', default="0")
parser_arg.add_argument("comment", type=str, nargs='?')
self.parsed_param_ref = parser_arg.parse_args(pom_param_list)
```

Kód 4.2: Ukázka nastavení argparse modulu pro BeakerLib příkaz rlRun.

### 4.1.3 Proces analýzy dat ze zdrojových souborů

Pro celý proces získání dat ze zdrojového souboru by se velmi dobře dal použit překladač pro *shell*. Nicméně tvorba tohoto překladače by byla velmi náročná a předmětem této práce není tvorba takového překladače, ale generování dokumentace. Z tohoto důvodu zde nebude použit, ale bude zde vysvětlen postup, který ho nahradil.

Generátor nejprve přečte celý zdrojový test po řádcích a jednotlivé příkazy rozdělí do tříd. Proces generování dokumentace řídí vytvořená třída *Parser*. Navíc obsahuje třídy, do kterých je rozdělen zdrojový kód. Tyto třídy slouží jako kontejnery a odpovídají *BeakerLib* fázi. Následně jsou tyto kontejnery využity pro celý proces generování dokumentace. Jedná se o dva typy a to `PhaseOutside` a `PhaseContainer`. Každá instance třídy `PhaseContainer` odpovídá jedné fázi *BeakerLib* testu a také obsahuje jméno této fáze, takže lze jednoduše poznat, o jakou fázi se jedná. Třída `PhaseOutside` pak obsahuje všechny příkazy, které se nachází mimo *BeakerLib* fáze. Navíc za pomoci modulu *shlex* budou vyhledávány všechny proměnné v testu, které se nachází mimo fáze. Následně se pokusí získat z těchto proměn-

ných klíčová slova testu.

V dalším kroku se v objektu hlavní třídy *Parser* spustí metoda `{get_doc_data()`. Tato metoda má za úkol projít všechny kontejnery fází a spustit v nich metodu `search_data()`, která vytvoří třídu `StatementDataSearcher()` a spustí její metodu pro získání dat z příkazů. Tato metoda očekává jako parametr jeden příkaz. V tomto příkazu se nejprve pokusí nahradit všechny existující proměnné za jejich hodnoty a poté daný příkaz rozdělí pomocí modulu *shlex*. Toto rozdělení je zde velice důležité, protože modul *argparse* umí pracovat pouze s polem dat, a proto musí být každý příkaz takto rozdělen, aby byl poté rozpoznán modulem *argparse*. Dalším krokem této metody je, že se pokusí identifikovat *BeakerLib* příkaz pomocí jeho jména. Pokud bude příkaz rozpoznán, spouští se ihned příslušná metoda, pro extrahování dat. Tyto metody, které extrahují data, jsou vytvořeny speciálně pro každý *BeakerLib* příkaz zvlášť a obsahují specificky nastavený modul *argparse* pro extrakci dat. Po získání dat se vytvoří *argparse* objekt, který obsahuje tato data. Na závěr jsou tyto objekty uloženy do daných kontejnerů. Pokud příkaz nebude rozpoznán, tak se opět spustí patřičná metoda. Nicméně navíc se zde také hledají možné proměnné, jako v případě třídy `PhaseOutside`.

Závěrem tohoto procesu je stav, kdy jednotlivé kontejnery obsahují *argparse* objekty se získanými daty. Dalším krokem pak je reprezentace získaných dat jako informace v přirozeném jazyce.

## 4.2 Generování informace v přirozeném jazyce

Tato sekce se zabývá přetvořením dat do informace v přirozeném jazyce.

### 4.2.1 Návrh generování informace v přirozeném jazyce

Po ukončení procesu popsaného v sekci 4.1.3, zmíněné kontejnery obsahují všechna potřebná data pro tvorbu informace. Nicméně, jak je přetvořit? Existují dva způsoby, jak tato data transformovat do informace v přirozeném jazyce.

První způsob je zaměřen na znalost *BeakerLib* příkazů. Jelikož se jedná o dokumentační generátor z *BeakerLib* testů, tak pokud se zaměříme pouze na tyto příkazy, tak lze tato data přetvořit do informací v přirozeném jazyce.

Druhou cestou může být vytvoření systému pro generování přirozeného jazyka, který bude zpracovávat tato data a díky znalosti anglického jazyka tato data přetvoří do informací v přirozeném jazyce. Výhodou tohoto systému je, že pracuje pouze s daty a není tedy závislý na znalosti jednotlivých příkazů. Tento systém je více rozebírán v sekci 4.2.2 Díky této vlastnosti by byl generátor v budoucnosti velmi dobře rozšiřitelný o další příkazy mimo *BeakerLib*. Jednalo by se pouze o upravení předchozí analýzy dat, aby popisovaný generátor dokumentace rozpoznal tyto nové příkazy a extrahoval z nich potřebná data.

Z těchto dvou způsobů byl vybrán první. Bohužel nemohla být vybrána implementace systému pro generování přirozeného jazyka, protože časová náročnost na implementaci daleko převyšuje výhodu získanou při použití tohoto způsobu.

### 4.2.2 Systém pro generování přirozeného jazyka

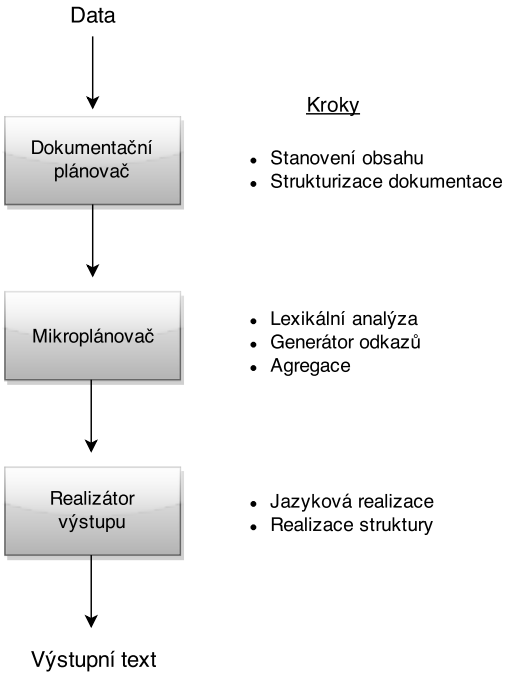
Návrh většiny systému pro generování přirozeného jazyka vychází z knihy pánů Reitera a Dalea [9], kde je popsaná celá architektura a postup vytvoření těchto systémů. Pro rychlé seznámení s tímto návrhem existuje článek v práci od pánů McBurneyho a McMillana [5].

Na obrázku 4.1 můžeme vidět návrh systému pro generování přirozeného jazyka. Návrh systému pro generování přirozeného jazyka se skládá ze tří hlavních komponent, které obsahují další kroky.

První komponenta se nazývá *Dokumentační plánovač*. Vstupem této části je množina dat, která musí být předána čtenáři dokumentace. Prvním krokem je *Stanovení obsahu*. V tomto kroku se data přepracují na zprávy. Zprávy jsou pokročilejší reprezentace vstupních dat. Následně přichází na řadu druhý krok *Strukturizace dokumentace*, který se pokusí tyto zprávy rozdělit do vět, které jsou srozumitelné pro člověka.

Druhá komponenta *Mikroplánovač* rozhoduje, co za slova budou použita pro popis každé zprávy. V *lexikálním analyzátoru* jsou přiřazená určitá slova jako součásti frází o každé zprávě. Poté se identifikují předměty a slovesa pro danou zprávu spolu s přídavnými jmény a příslovci. Následující dva kroky zjemňují tyto fráze z důvodu lepší čitelnosti.

Poslední komponenta *Realizátor výstupu* generuje věty v přirozeném jazyce z předchozích frází. Tato komponenta obsahuje pravidla pro tvorbu vět, které obsahují části řeči a slov obdržených z *Mikroplánovače*. Tyto věty představují výstupní text.




Obrázek 4.1: Návrh Nature Language Generation systemu z knihy od Reitera a Dalea [9]

### 4.2.3 Rozšiřitelnost generátoru o příkazy mimo BeakerLib

V předchozí sekci 4.2.1 bylo rozhodnuto pro generování informace na základě podrobné znalosti všech *BeakerLib* příkazů, na místo tvorby systému pro generování přirozeného jazyka. Tento přístup má velkou nevýhodu. Při jeho použití se velmi snižuje budoucí rozšiřitelnost celého generátoru o příkazy mimo *BeakerLib*. Nicméně nelze říci, že při použití tohoto přístupu se stane z generátoru absolutně nerozšiřitelný nástroj, ale pouze aplikace nových

příkazů bude daleko náročnější než u druhého zmiňovaného přístupu.

Kvůli tomuto poznatku byl generátor vylepšen pomocí třídy `DocumentationInformation`, která reprezentuje extrahovaná data. Tato třída obsahuje čtyři instanční proměnné a to *topic*, *importance*, *action* a *options*. Z těchto čtyř proměnných budou blíže popsány dvě. Instanční proměnná *topic* obsahuje informaci, která reprezentuje fakt, ke kterému se jednotlivý příkaz pojí. Například u příkazu `rlRun`, který spouští zvolený příkaz, se jedná o fakt *command*, u příkazu `rlFileExists`, který kontroluje, zda soubor existuje, je fakt *file*. Druhá proměnná *action* obsahuje slovo, vyjadřující činnost celého příkazu. Například u příkazu `rlFileBackup`, který zálohuje zvolené soubory, bude tato instanční proměnná obsahovat slovo *backup*. Díky třídě `DocumentationInformation` bylo možné popsat každý příkaz a zvýšit budoucí rozšiřitelnost generátoru o příkazy mimo *BeakerLib*.

### 4.2.4 Proces generování informace

Způsobem popsaným v sekci 4.1.3 byly kontejnery odpovídající jednotlivým fázím *BeakerLib* testů naplněny *argparse* objekty s extrahovanými daty. V dalším kroku popsaném v této sekci se opět z hlavní třídy `Parser` spustí metoda s názvem `get_documentation_information()`, jejímž cílem je opět projít jednotlivé kontejnery a v nich za pomocí metody `translate_data` vytvoří instanci třídy `DocumentationTranslator`.

Objekt třídy `DocumentationTranslator` obsahuje metodu, která zpracovává `argparse` objekty a reprezentuje je jako instanci třídy `DocumentationInformation` zmiňované v předchozí sekci 4.2.3. V tomto bodě se také ukládá důležitost jednotlivých příkazů do instance třídy *documentation_information*. Tato důležitost je tvořena na základě subjektivního porozumění `BeakerLib` příkazům a tyto příkazy mohou nabývat hodnot od jedné do pěti, kde jedna je nejméně důležitý příkaz a naopak pět je zase nejdůležitější příkaz.

Následující krok se zabývá transformací `DocumentationInformation` objektů do informací v lidské řeči. K tomuto účelu bylo vytvořeno mnoho malých tříd, které odpovídají všem kombinacím instančních proměnných `topic` a `action` z třídy `DocumentationInformation`, jejichž objekty obsahují instanční proměnnou s daty reprezentujícími informaci v lidské řeči. Celou transformaci z objektu `DocumentationInformation` do připravených malých objektů má na starosti třída `GetInformation`. Tento objekt obsahuje dvourozměrné pole, které se velmi podobá precedenční tabulce ze syntaktické analýzy s tím rozdílem, že sloupce odpovídají hodnotám v proměnné `topic` a řádky odpovídají hodnotám v proměnné `action`. Navíc toto pole místo znaků $<$, $>$ a $=$ obsahuje zmíněné malé objekty. Díky znakům $<$, $>$ a $=$ se v precedenční tabulce řídí syntaktická analýza výrazů. Nicméně v tomto případě byly zmíněné znaky nahrazeny za malé objekty z toho důvodu, že není potřeba kontrolovat pokročilé pravidla syntaktické analýzy. Zmíněné pole, ve velmi zmenšené podobě, můžeme vidět na ukázce 3.

Proces převodu objektu `DocumentationInformation` do malých objektů uchovávající informaci v přirozeném jazyce je uskutečněn v metodě objektu `GetInformation`. Algoritmus řídící tento proces můžeme vidět níže 4.3. Algoritmus začíná vyhledáním patřičného indexu řádku a sloupce z dat v poskytnutém objektu `DocumentationInformation`, pomocí metod `get_topic_number()` a `get_action_number()`. Následně je získána hodnota, která se nachází ve dříve zmíněném dvourozměrném poli a odpovídajícím získaným indexům řádku a sloupce. Dále algoritmus zjišťuje, zda získaná hodnota je prázdná. Pokud se hodnota rovná nule, tak neexistuje informace v přirozeném jazyce, která by popsala danou kombinaci ze zjištěného řádku a sloupce. V druhém případě hodnota obsahuje odkaz na malou třídu a po zjištění, že hodnota není nulová se vytvoří objekt zmíněné třídy. Tento

algoritmus poté vrací buď objekt s informací v přirozeném jazyce a nebo prázdný řetězec.

```
array = [
# topic: FILE(DIRECTORY),    STRING            PACKAGE       # ACTIONS
[InformationFileExists,      0,    InformationPackageExists ],# exists
[InformationFileNotExists,   0, InformationPackageNotExists ],# !exists
[InformationFileContain,     0,                  0          ],# contain
[InformationFileNotContain,  0,                  0          ],# !contain
[InformationFilePrint,       0,    InformationPackagePrint  ],# print
[InformationFileResolve,     0,                  0          ],# resolve
[InformationFileCreate, InformationStringCreate, 0          ],# create
[InformationFileCheck,       0,     InformationPackageCheck ] # check
```

Kód 3: Ukázka dvourozměrného pole z třídy GetInformation.

```
1  def get_information_from_facts(self, information_obj):
2    information = ""
3    topic = information_obj.get_topic()
4    for action in information_obj.get_action():
5      column = self.get_topic_number(topic)
6      row = self.get_action_number(action)
7      information_class = self.array[row][column]
8      if information_class:
9        information = information_class(information_obj)
10       information.set_information()
11   return information
```

Kód 4.3: Algoritmus řídící proces převodu objektu DocumentationInformation do informací v přirozeném jazyce.

Závěrem procesu generování informace jsou v kontejneru fází uloženy objekty s informacemi. Nicméně v této podobě je složité jednotlivé informace v přirozené řeči měnit. Z tohoto důvodu by bylo v budoucnu lepší tyto malé objekty s informacemi generovat z konfiguračního souboru, který by měl snadnější úpravu.

## 4.3 Vytvořená dokumentace z generátoru

Závěrečným krokem navrženého postupu generování dokumentace je vygenerování dokumentace. Na ukázce 2 můžeme vidět vygenerovanou dokumentaci k *apache* testu, který je na ukázce 2.1. Můžeme vidět, že dokumentace poskytuje informace vycházející z *BeakerLib* příkazů a čtenář si může představit co má daný test dělat. Následně si můžeme všimnout, že v dokumentaci jsou informace, které říkají, že příkaz musí skončit úspěšně nebo neúspěšně. Tyto příkazy jsou obsaženy v příkazu `rlRun` a jedná se o *shellové* příkazy, kterým generátor nerozumí. Pro vygenerování lepší informace by bylo nutné implementovat překladač *shellových* příkazů, neboť s větším počtem použitých *shellových* příkazů, klesá důvěryhodnost celé dokumentace. Jedná se tedy o další možná rozšíření. Dále si můžeme všimnout, že chybí jméno autora testu, popis a klíčová slova. Opět se jedná o možné rozšíření generátoru. V této části se nachází nová informace, která nebyla v původním návrhu. Tato informace

se jmenuje *Test environmental variables*. Byla do návrhu přidána až v době implementace generátoru z důvodu potřeby testerů, kdy potřebují vědět, zda v testu existují environmentální proměnné. Následující rozšíření generování dokumentace je vytvoření šablon, do kterých se budou jednotlivé informace vkládat. Mohlo by se jednat o naformátované *HTML* soubory a nebo *PDF* soubory.

```
Test launch: apache-test.sh
Test environmental variables: -

Setup "Setup" [ Unknown commands: 1, Total: 7 ]
  Package httpd must be installed
  Command "TmpDir=$(mktemp -d)" must run successfully
  Files or directories /var/www/html and /var/log/httpd are backed up
  Command "echo 'Welcome to Test Page!' > /var/www/html/index.html"
    must run successfully
  Command "rm -f /var/log/httpd/*" must run successfully
  Service(s): httpd must be running

Test "Test Existing Page" [ Unknown commands: 0, Total: 5 ]
  Command "wget http://localhost/" must run successfully
  File(directory): "index.html" must exist
  File: "index.html" must contain pattern: "Welcome to Test Page"
  File: "/var/log/httpd/access_log" must contain pattern:
     "GET / HTTP.*200"

Test "Test Missing Page" [ Unknown commands: 0, Total: 5 ]
  Command "wget http://localhost/missing.html 2>stderr" exit code
    must match: 1,8
   File(directory): "missing.html" must not exist
   File: "stderr" must contain pattern: "Not Found"
   File: "/var/log/httpd/access_log" must contain pattern:
     "GET /missing.html HTTP.*404"
   File: "/var/log/httpd/error_log" must contain pattern:
     "does not exist.*missing.html"

Cleanup "Cleanup" [ Unknown commands: 1, Total: 4 ]
  Command "rm -r $(mktemp -d)" must run successfully
  Restoring backed up files to their original location
  Service(s) httpd will be restored into original state
```

Dokumentace 2: Ukázka vygenerované dokumentace k apache testu.

## 4.4 Zkracování výsledné dokumentace

Z předchozí části víme, že generátor spolehlivě tiskne informace v lidské řeči z nasbíraných dat. Nicméně se zde vyskytuje jeden velký problém, a to je velikost výstupní dokumentace. Jedná se o to, že *BeakerLib* testy mohou být velmi velké a k takto velkým testům výše

uvedený postup vygeneruje velmi rozsáhlou dokumentaci. Příliš rozsáhlá dokumentace, ale typicky není pro čtenáře přijatelná. Je tedy zapotřebí dokumentaci nějakým způsobem zkrátit.

Prvním způsobem je spojování informací. Na ukázce 2 ve fázi `Setup` můžeme vidět dvě informace týkající se balíčku `httpd`. Tyto dvě informace by se mohly spojit v jednu více přiblížené lidské řeči. V tomto případě by se informace: *Package httpd must be installed* a *Service(s): httpd must be running* mohly spojit do následující: *Package httpd must be installed and running*. Díky tomuto spojení se zvýší důležitost informace. Nicméně tento způsob ještě není implementovaný.

Následující způsob se týká zkrácení věty samotné. V dokumentaci se může objevit například tato informace „*Shows a message about package: mysql-server firefox.x86_64 httpd tigervnc-server-minimal perl-Test-WWW-Selenium version.*“. Na této informaci si můžeme všimnout, že obsahuje velké množství balíčků. Tuto informaci lze zkrátit na poloviční velikost pokud většinu balíčků nahradíme za ”...”. Zkrácená informace by mohla vypadat následovně: „*Shows a message about package: mysql-server, httpd, ... version.*“. Celá informace by se mohla objevit na konci dokumentace, kdyby čtenář potřeboval vědět celou délku informace. Nicméně tento způsob ještě také není implementovaný.

Posledním způsobem zkracování je smazání jednotlivých informací, aby se celkový počet vešel na stranu A4. Nicméně u tohoto způsobu vyvstává problém, kterou informaci smazat a jakou zase ponechat. Tento problém se nazývá ”problém batohu”a podrobněji je vysvětlen v sekci 4.4.1. Tento způsob je v generátoru implementován. Pomocí přepínače `-s value` lze nastavit velikost výsledné dokumentace. Přepínač `-s` má nastavenou standardní velikost na třicet dva řádků, které odpovídají velikosti A4 strany. Generátor maže pouze informace z fází *BeakerLib* testu, které budou uvedené ve zmíněné prostřední části. Následující ukázka 3 zobrazuje *apache* test se třinácti informacemi. Na této ukázce můžeme vidět, že se odstranily méně důležité informace a zvýšil se počet nerozpoznaných příkazů o proti původní ukázce 2.

### 4.4.1 Problém batohu

Tento problém je velmi dobře popsán následujícím příběhem. Lupič se vkrade do domu a najde obrovskou kořist. Nicméně lupič takový lup nečekal a vzal si jen malý batoh, který unese jen zboží do určité váhy nebo se pak roztrhne. V domě je také mnoho zboží o různé váze a ceně. Lupič musí vybrat takovou kombinaci, která mu přinese co největší zisk. V tomto případě si lze batoh představit jako stanu A4 s váhou třiceti dvou řádků. A zboží jako informace v přirozeném jazyce s cenou, která odpovídá důležitosti informace a váhou, která odpovídá počtu řádků, na kterých bude informace v dokumentaci zobrazena.

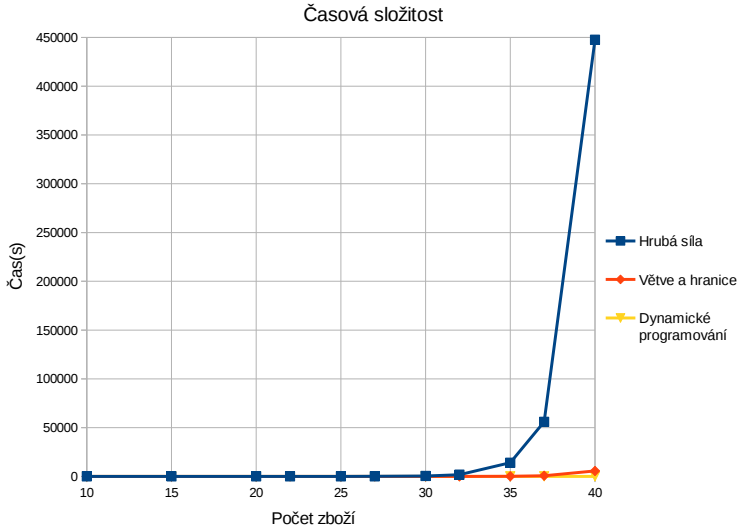
Pro vyřešení tohoto problému bylo uvažováno mezi třemi metodami. První je metoda ”Hrubou silou”. Tato metoda prochází všechny kombinace a v každém průchodu vypočte jejich celkovou hmotnost a cenu. Pokud je hmotnost nižší než nosnost batohu a zároveň jejich cena vyšší než dosud nejvyšší nalezená, označí se dané řešení jako řešení problému, které je po průchodu všemi kombinacemi vypsáno [12].

Druhým způsobem je metoda větví a hranic. Algoritmus je založen na řešení hrubou silou (první způsob). Rozdíl je v tom, že větve, které už nemůžou být lepší než dosavadní řešení, nebudou prozkoumávány.

Poslední metoda využívá užití dynamického programování. Základem této metody je dvourozměrné pole, které tvoří tabulku. V této tabulce jsou na jedné ose jednotlivé předměty a na druhé všechny přípustné ceny batohu (horní hranice této osy je tedy součet všech

```
Test launch: apache-test.sh
Test environmental variables: -

Setup "Setup" [ Unknown commands: 3, Total: 7 ]
  Package httpd must be installed
  Command "TmpDir=$(mktemp -d)" must run successfully
  Files or directories /var/www/html and /var/log/httpd are backed up
  Command "rm -f /var/log/httpd/*" must run successfully

Test "Test Existing Page" [ Unknown commands: 1, Total: 5 ]
  Command "wget http://localhost/" must run successfully
  File(directory): "index.html" must exist
  File: "index.html" must contain pattern: "Welcome to Test Page"
  File: "/var/log/httpd/access_log" must contain pattern: "GET /
    HTTP.*200"

Test "Test Missing Page" [ Unknown commands: 2, Total: 5 ]
  Command "wget http://localhost/missing.html 2>stderr" exit code
     must match: 1,8
  File(directory): "missing.html" must not exist
  File: "stderr" must contain pattern: "Not Found"

Cleanup "Cleanup" [ Unknown commands: 2, Total: 4 ]
  Command "rm -r $(mktemp -d)" must run successfully
  Restores backed up files to their original location
```

Dokumentace 3: Ukázka vygenerované dokumentace k apache testu pro třináct informací.




Obrázek 4.2: Časová složitost tří metod na řešení problému batohu.

cen předmětů dané instance problému). Obsahem této tabulky jsou poté váhy. Algoritmus prochází všemi předměty a jejich kombinací získává nejlepší cenu.

Na grafu 4.2 můžeme vidět, že nejhorší časovou složitost má metoda „Hrubá síla“. Nejlepší naopak má metoda dynamické programování. Data zobrazená v tomto grafu byla získána z práce pana Marka Handla [2]. Díky získaným výsledkům jsem se rozhodl implementovat metodu dynamické programování pro řešení problému batohu. Algoritmus byl získán z internetového článku, zabývající se metodou dynamického programování pro různé programovací jazyky [11].

# Kapitola 5

# Experimenty a testování

V této kapitole budou zobrazeny výsledky dokumentačního generátoru. Následně kvalita vygenerované dokumentace bude zhodnocena na deseti náhodně zvolených testech.

## 5.1 Porovnání ručně a automaticky tvořených dokumentací

Před implementací generátoru byly vybrány tři testy o různých velikostech a k těmto testům byly ručně vytvořeny ideální dokumentace. V této sekci bude probíhat porovnávání těchto tří ručně vytvořených dokumentací s dokumentacemi vygenerovanými vytvořeným generátorem.

### 5.1.1 pxe-boot test

Prvním testem je *pxe-boot* test, který je nejmenší z vybraných. Na ukázce 4 je ručně vytvořená dokumentace a 5 je vygenerovaná vytvořeným generátorem. Na první pohled je vidět, že v první části chybí popis testu, jméno autora, klíčová slovy atd. Tato část je zmíněna v předchozí kapitole a jedná se o možné rozšíření.

```
Description: Syslog testing
Author:  Alexander Todorov <atodorov@redhat.com>
Keywords: RedHatEnterpriseLinux7, RedHatEnterpriseLinux*, Fedora*

Test structure: 1xTest

Test launch: ./pxe-boot-test.sh

Test PXE-Boot [Unknown commands = 2, Total = 16]
Testing of syslog in OS made by RedHat

Expected result:
```

Dokumentace 4: Ukázka ručně vytvořené dokumentace k pxe-boot testu

Velký rozdíl se nachází v prostřední části, která je velmi rozdílná. Můžeme vidět, že vygenerovaná dokumentace neposkytuje užitečnou informaci. Příčinou špatného výsledku

vygenerované dokumentace je ta, že *pxe-boot* test obsahuje velké množství *shellových* příkazů. Tento fakt lze vypozorovat z počtu nerozpoznaných příkazů, který má hodnotu čtrnáct z šestnácti. Zde je také vidět důležitost informace o důvěryhodnosti vygenerované dokumentace.

```
Test launch: pxe-boot-test.sh
Test environmental variables: FAMILY

Test PXE-Boot [ Unknown commands: 14, Total: 16 ]
  Message "Unknown syslog file!" will be created in to log
  Value \$? must be 0
```

Dokumentace 5: Ukázka vygenerované dokumentace k pxe-boot testu

Závěrem lze říci, že vygenerovaná dokumentace neposkytuje důležité informace a spíše se jedná o zavádějící informace. Nicméně tato dokumentace upozorňuje na svou kvalitu vysokým počtem nerozpoznaných příkazů.

### 5.1.2 apache test

Tento test byl použit pro případné ukázky. Ručně vytvořenou dokumentaci k tomuto testu lze vidět v sekci zabývající se finálním formátem dokumentace 1. Vygenerovaná dokumentace k tomuto testu je na ukázce 2 v sekci zabývající se tisknutím dokumentace. Tyto dvě dokumentace nemají příliš mnoho rozdílů a dokumentace vytvořená generátorem poskytuje velmi podobné informace, které lze pozorovat v ideální dokumentaci k tomuto testu. V tomto případě generátor vytvořil dokumentaci poskytující důležité informace.

### 5.1.3 mozila test

Posledním z testů je *mozila* test, který je z vybraných testů největší. Na ukázce 6 lze postřehnout ideální ručně vytvořenou dokumentaci a na ukázce 7 můžeme vidět vygenerovanou dokumentaci. Z vygenerované dokumentace lze vypozorovat, že *mozila* test opět obsahuje velké množství *shell* příkazů. Můžeme vidět, že vygenerovaná dokumentace obsahuje užitečné informace, ale i hodně informací, kterým čtenář porozumí pokud rozumí *shellovým* příkazům. Navíc si můžeme všimnout, že v tomto testu jsou použity dvě environmentální proměnné.

### 5.1.4 Závěr porovnávání

V této sekci jsme viděli porovnávání dokumentací k třem testům. V příkladu s *apache* testem generátor vytvořil užitečnou dokumentaci. Ve zbylých dvou příkladech bylo vidět, že s větším počtem *shellových* příkazů klesá kvalita výstupní dokumentace. Pro zlepšení výsledků je potřeba více rozumět *shellovým* příkazům.

```
Description:  This test is for testing Bugzilla version 3.6.
Author:  Dave Lawrence <dkl@redhat.com>
Keywords: mysql, httpd, Bugzilla
Test structure: Setup, Test, Cleanup
Test launch: ./mozila-test.sh

Setup [Unknown commands = 2, Total = 17]
Starting mysqld and httpd service
Setting up the Bugzilla web root directory and Bugzilla configuration
files
Installing required packages

Test [Unknown commands = 1, Total = 7]
Running common sanity tests, Selenium tests and WebService tests
Creating test data
Printing test results

Cleanup [Unknown commands = 0, Total = 2]
Stoping httpd and mysqld service
```

Dokumentace 6: Ukázka ručně vytvořené dokumentace k mozila testu

## 5.2 Vygenerované dokumentace k deseti náhodně zvoleným testům

Pro zhodnocení kvality generátoru bylo vybráno deset testů, ke kterým byla vygenerována dokumentace. Při generování dokumentace výše popsaným generátorem došlo k chybám z důvodu špatného získávání proměnných ze zdrojového kódu. Tento problém lze pouze řešit pomocí překladače pro *shell*. Generátor upozorňuje na vzniklé chyby pomocí chybových hlášení. Vytvořené dokumentace byly poté předány Mgr. Davidu Kutálkovi z firmy Red Hat Czech na zhodnocení. Od pana Mgr. Davida Kutálka bylo obdrženo následující zhodnocení: „Celkově vzato je generátor docela použitelný - většina testů byla zdokumentována tak, že bylo patrné, k čemu test slouží. Je stále co zlepšovat, mimojiné se pokusit zpracovat cykly a větvení, funkce a zvážit uvedení některých rlLog komentářů“. Z výše uvedeného zhodnocení plyne, že generátor vygeneroval použitelnou dokumentaci k deseti vybraným testům. Nicméně pro poskytování informací o cyklech, podmínkách a funkcích je zapotřebí implementovat překladač pro *shell*. Jednotlivé hodnocení k testům můžeme najít v příloze C.

```
Test launch: mozila-test.sh [VARIABLE]
Test environmental variables: BEAKERLIB_DIR, OUTPUTFILE

Setup [ Unknown commands: 6, Total: 24 ]
  Command "tar zxvf bugzilla.tar.gz" must run successfully
  Command "yum -y install httpd mysql-server firefox.x86_64
    tigervnc-server-minimal perl-Test-WWW-Selenium
    java-1.6.0-openjdk" must run successfully
  Package $i must be installed
  Command "yum -y remove firefox.i386" exit code must match: 0,1
  Command "rm -rf /var/www/html/bugzilla" must run successfully
  Command "mv bugzilla /var/www/html/bugzilla" must run successfully
  Command "cd /var/www/html/bugzilla" must run successfully
  Command "mkdir -p data" must run successfully
  Command "cp t/config/params data/." must run successfully
  Starts service mysqld
  Command "echo 'set global max_allowed_packet=1000000000000;' | mysql"
    must run successfully
  Command "cat t/config/create_bugs_user.sql | mysql" must run successfully
  Command "cp t/config/bugzilla.conf /etc/httpd/conf.d/" must run successfully
  Starts service httpd
  Command "Xvnc :4 -alwaysshared -geometry 1600x1200 -depth 24 -SecurityTypes
    None 2>/dev/null >/dev/null &" must run successfully
  Command "env DISPLAY=:4 java -jar t/config/selenium-server.jar -log
    selenium.log 2>/dev/null >/dev/null &" must run successfully
  File(directory): "$( rpm -ql firefox | grep /usr/lib\(64\)\?/
    firefox-[^/]\+/firefox )" must exist
  Command "sed -i 's|FIREFOX_PATH|\$( rpm -ql firefox | grep /usr/lib\(64\)
    \?/firefox-[^/]\+/firefox )|g' t/config/selenium_test.conf" must run
    successfully

Test [ Unknown commands: 17, Total: 21 ]
  Command "cd /var/www/html/bugzilla" must run successfully
  Reports test "Overall testing result" with result PASS

Cleanup [ Unknown commands: 0, Total: 2 ]
  Kills service httpd
  Kills service mysqld
```

Dokumentace 7: Ukázka vygenerované dokumentace k mozila testu

# Kapitola 6

# Závěr

Cílem této bakalářské práce bylo především navrhnout, vytvořit a ověřit generátor dokumentace z *BeakerLib* testů, který bude vytvářet užitečnou dokumentaci v lidské řeči.

Na počátku byl vysvětlen vznik dokumentace pro *BeakerLib* testy. Tento návrh byl poté diskutován a schválen testery ze společnosti Red Hat Czech.

Tato práce byla prezentována na mezinárodní konferenci Devconf a také se účastnila Excel@FIT konference.

Na ukázce 2 můžeme vidět, že výsledná dokumentace obsahuje užitečné informace. Tohoto výsledku bylo dosaženo statickou analýzou *BeakerLib* příkazů. Následně získaná data jsou přetvořena do informací v lidské řeči, pomocí znalosti všech *BeakerLib* příkazů. Na závěr jsou tyto informace vytištěny do výsledné podoby. Kvalita vygenerované dokumentace se odvíjí od počtu použitých *shellových* příkazů. S větším počtem těchto příkazů může dokumentace poskytovat neužitečné informace. Nicméně porozumění *shellovým* příkazům nebylo primárním cílem této práce, ale k zlepšení výstupní dokumentace by bylo třeba přidat implementaci těchto příkazů.

Navíc se podařilo implementovat řešení pro tzv. *problém batohu*, který vznikl z důvodu potřeby snížení velikosti dokumentace.

Celý proces tvorby generátoru byl řízen metodikou *programování řízené testy*. Díky této metodice byly otestovány všechny části a generátor poskytuje vysokou funkčnost. Vytvořené testy lze najít na přiloženém CD. Zároveň funkčnost po celou dobu tvorby generátoru byla testována na třech vybraných testech. K těmto testům byly ručně vytvořeny dokumentace, které byly poté porovnávány s automaticky vygenerovanými dokumentacemi. Následně byl generátor otestován na deseti náhodně zvolených testech a zhodnoceny zaměstnanci z firmy Red Hat Czech. Z těchto hodnocení plyne, že generátor vytváří použitelné dokumentace s dodatečnými informacemi o spuštění testu nebo o environmentálních proměnných testu.

Výhodou tohoto generátoru je okamžité vytváření dokumentace z testů. Navíc dokumentace obsahují důležitou informaci ohledně důvěryhodnosti celé dokumentace. Dokumentace navíc poskytuje informaci o environmentálních proměnných a počtu parametrů, nutných pro spuštění testu.

Dalším důležitým rozšířením je získávání klíčových slov z testu. Počáteční sběr dat je implementován, ale je potřeba tato data sbírat i z jednotlivých příkazů. Následně zajímavým problémem je generování shrnutí testu. Generátor by měl porozumět jednotlivým fázím a poté vygenerovat souhrnnou informaci o celém testu.

Cestou pro získání dalších důležitých dat může být dynamická analýza celého testu. Zde by mohlo být jednodušší zjistit počet potřebných parametrů pro spuštění testů a také zisk dodatečných informací o použitých knihovnách. Pro testery by poté dokumentace mohla

poskytovat informaci o chybě v testu a případnou radu, jak tuto chybu vyřešit.

Mezi další rozšíření této práce patří úprava informace o důvěryhodnosti dokumentace, spojování informací do jedné, vytvoření šablon dokumentace (pdf, html, atd.), lépe reprezentovat větší množství dat v informaci (větší počet souborů nebo balíčků).

# Literatura


[1] Chan, W.-K.; Cheng, H.; Lo, D.: Searching Connected API Subgraph via Text Phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1614-9, s. 10:1–10:11, doi:10.1145/2393596.2393606.

[2] Handl, M.: Problém batohu metodami, branch and bound, dynamické programování, heuristika s testem [online]. `http://www.mareen.cz/school/x36paa/03-batohB_B/03-batohB_B.pdf`.

[3] Heesch, D.: Doxygen. `http://www.stack.nl/~dimitri/doxygen/`.

[4] Inoue, K.; Yokomori, R.; Fujiwara, H.; aj.: Component rank: relative significance rank for software component search. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, May 2003, ISSN 0270-5257, s. 14–24, doi:10.1109/ICSE.2003.1201184.

[5] McBurney, P. W.; McMillan, C.: Automatic Documentation Generation via Source Code Summarization of Method Context[online]. `http://www.cse.nd.edu/ cmc/papers/mcburney_icpc_2014.pdf`, 2013-04-02 [cit. 2015-02-15].

[6] McMillan, C.; Grechanik, M.; Poshyvanyk, D.; aj.: Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0445-0, s. 111–120, doi:10.1145/1985793.1985809.

[7] Muller, P.; Hudlicky, O.; Hutar, J.; aj.: BeakerLib. `https://fedorahosted.org/beakerlib/`, 2013-04-02 [cit. 2015-02-15].

[8] Oracle Corporation: Javadoc. `http://www.naturaldocs.org/`.

[9] Reiter, E.; Dale, R.: *Building Natural Language Generation Systems*. Cambridge University Press, 2006, iSBN 9780521024518.

[10] Valure, G.: Natural Docs. `http://www.naturaldocs.org/`.

[11] WWW stránky: Knapsack problem/0-1. `http://rosettacode.org/wiki/Knapsack_problem/0-1`.

[12] WWW stránky: Problém batohu. `http://woq.nipax.cz/sobaka/01_problem_batohu.php`.

# Příloha A

# Obsah CD

**Adresářová struktura CD:**

- *BeakerLib_tests*: Složka s *BeakerLib* testy použitými pro testování generátoru.
  - *Three_base_tests*: Složka obsahující tři testy, které byly použity pro testování generátoru dokumentace při jeho vývoji.
  - *Ten_BeakerLib_tests_with_feedback*: Složka obsahující deset *BeakerLib* testů, které byly použity k závěrečnému testování a vyhodnocení kvality vygenerované dokumentace z generátoru. Složka také obsahuje zpětnou vazbu k těmto testům od zaměstnance z firmy Red Hat Czech.
- *Thesis*: Složka obsahující zdrojové soubory LaTeX a *PDF*.
- *Source_code*: Složka obsahující zdrojový soubor generátoru dokumentace.
- *Source_code_test*: Složka obsahující testovací soubor generátoru dokumentace.
- *Readme.pdf*: Soubor obsahující manuál k programu a popis obsahu CD.

# Příloha B

# Manuál

Jedná se generátor dokumentace z *BeakerLib* testů. Program je napsán v programovacím jazyce *Python*.

- *Instalace*: Stačí stáhnout program *documentation_generator* z přiloženého CD nebo stáhnout z Github repositáře[1] na pevný disk. Dále je potřeba mít nainstalovanou alespoň verzi *Python 2.7* nebo vyšší.
- *Spuštění generátoru dokumentace*: Skript lze spustit následujícím způsobem. Nejdříve musíme vybrat verzi *Python* překladače např. *Python 2.7* a pak za něj dopíšeme *documentation_generator.py*, což je výše popsaný generátor dokumentace. Celé zakončíme výběrem testu, ze kterého chceme vygenerovat dokumentaci. Dále lze použít následující přepínače:
  - *-s SIZE*: Přepínač pro nastavení velikosti výstupní dokumentace. *SIZE* značí počet řádku, na kterých budou zobrazeny informace z fází.
  - *-l*: Přepínač pro zobrazování informací z *rlLog* příkazů. Standardně jsou logovací zprávy zakázány, ale pomocí tohoto přepínače je lze povolit.
  - *-h*: Tisknutí nápovědy s použitím programu.

  Příklady spuštění můžeme vidět na následujíchich příkladech s výsledky:

[1] https://github.com/rh-lab-q/bkrdoc

```
> python2.7 documentation_generator.py mozila-test.sh -s 15
Test launch: mozila-test.sh [VARIABLE]
Test environmental variables: BEAKERLIB_DIR, OUTPUTFILE

Setup [ Unknown commands: 12, Total: 24 ]
  Command "tar zxvf bugzilla.tar.gz" must run successfully
  Package $i must be installed
  Command "yum -y remove firefox.i386" exit code must match: 0,1
  Command "rm -rf /var/www/html/bugzilla" must run successfully
  Command "mv bugzilla /var/www/html/bugzilla" must run successfully
  Command "cd /var/www/html/bugzilla" must run successfully
  Command "mkdir -p data" must run successfully
  Command "cp t/config/params data/." must run successfully
  Starts service mysqld
  Command "cat t/config/create_bugs_user.sql | mysql" must run successfully
  Command "cp t/config/bugzilla.conf /etc/httpd/conf.d/" must run successfully
  Starts service httpd

Test [ Unknown commands: 19, Total: 21 ]
  Command "cd /var/www/html/bugzilla" must run successfully
  Reports test "Overall testing result" with result FAIL

Cleanup [ Unknown commands: 1, Total: 2 ]
  Kills service httpd
```

Dokumentace 8: Ukázka spuštění generátoru dokumentace pro mozila test s přepínačem -s.

```
> python3.3 documentation_generator.py apache-test.sh
Test launch: apache-test.sh
Test environmental variables: -

Setup "Setup" [ Unknown commands: 1, Total: 7 ]
  Package httpd must be installed
  Command "TmpDir=$(mktemp -d)" must run successfully
  Files or directories /var/www/html and /var/log/httpd are backed up
  Command "echo 'Welcome to Test Page!' > /var/www/html/index.html"
    must run successfully
  Command "rm -f /var/log/httpd/*" must run successfully
  Service(s): httpd must be running

Test "Test Existing Page" [ Unknown commands: 0, Total: 5 ]
  Command "wget http://localhost/" must run successfully
  File(directory): "index.html" must exist
  File: "index.html" must contain pattern: "Welcome to Test Page"
  File: "/var/log/httpd/access_log" must contain pattern:
    "GET / HTTP.*200"

Test "Test Missing Page" [ Unknown commands: 0, Total: 5 ]
  Command "wget http://localhost/missing.html 2>stderr" exit code
    must match: 1,8
  File(directory): "missing.html" must not exist
  File: "stderr" must contain pattern: "Not Found"
  File: "/var/log/httpd/access_log" must contain pattern:
    "GET /missing.html HTTP.*404"
  File: "/var/log/httpd/error_log" must contain pattern:
    "does not exist.*missing.html"

Cleanup "Cleanup" [ Unknown commands: 1, Total: 4 ]
  Command "rm -r $(mktemp -d)" must run successfully
  Restoring backed up files to their original location
  Service(s) httpd will be restored into original state
```

Dokumentace 9: Ukázka spuštění generátoru dokumentace pro apache test.

# Příloha C

# Hodnocení deseti vygenerovaných dokumentací

Poznámky k bakalářské práci bkrdoc. Vygenerované testy lze najít na přiloženém CD nosiči.

## C.1 test1

Dokumentace je dobře čitelná, základní poslání a obsah testu je z ní zřejmé. Není zřejmý význam proměnné CONF_VALUE, která je průběžně nastavována voláním funkce, která je jen započítána do 'unknown commands'.

Za slabší považuji popis testovací fáze 'various argument types', kde jsou popsány dva příkazy, přičemž jeden je volán v cyklu a druhý nikoliv. Bylo by dobré se např. dozvědět, jakých hodnot proměnna arg v cyklu nabývá, tj. jaké parametry jsou testovány.

## C.2 test2

Obsahově dokumentace obsahuje vše potřebné. Při čtení mne trochu zarazila substituce proměnné TmpDir, neboť to vypadá, jako by test v Cleanup fázi znovu vytvářel pracovní adresář. Naopak substituce proměnné AVCPROBLEMDIR jsem si na první pohled nevšiml a vypadá vhodně.

## C.3 test3

Dobře zdokumentovaný test. Drobnou chybou je nedokonalé nahrazení proměnné DIR_DELETE v poslední testovací fázi (v dokumentaci zůstal kousek jména proměnné - _DELETE).

## C.4 test4

Dokumentace dostačující. Pokud by postihla i volané funkce bylo by zřejmější, že během poslouchání na sběrnici dbus je vyvolán pád (např. 'Calls a function generate_crash').

## C.5 test5

Z dokumentace k tomuto testu jsem nebyl schopný pochopit, co dělá. V případě doplnění o volané funkce a komentáře logované pomocí rlLog by to zřejmě bylo lepší.

## C.6 test6

Z dokumentace mi není jasné, co test dělá. Není nijak zohledněno podmínečné větvení testovacího kódu - jsou vypsány příkazy z obou větví bez jejich rozlišení.

## C.7 test7

Dokumentace se mi jeví celkem pochopitelná. Drobná chybka v parsování víceřádkových řetězců. Také se zdá, že použití proměnných formou $ není nahrazováno.

## C.8 test8

Z dokumentace je mi zřejmé, co test samotný dělá. Jen sekce Setup je trochu neutěšená bez jediné poznámky.

## C.9 test9

Z dokumentace jsem přibližně pochopil smysl testu. Test samotný je však složitější a není moc komentovaný. Větvení opět není ošetřeno.

## C.10 test10

Testované příkazy jsou v generované dokumentaci uvedené. Bylo by dobré, kdyby z dokumentace bylo vidět, že test obsahuje a spouští vlastní server, proti kterému přikazy zkouší.